ALU Firmware Development Plan

#### Dillon Capalongo

# Scope:

This project will develop an Arithmetic Logic Unit (ALU) to command a system that performs error detection, location, and correction in air traffic control. Future researchers involved in air traffic communications will apply this research to correct errors using Cyclic Redundancy Check (CRC). This will improve the efficiency of secondary surveillance radar involved in data transferring, which is key to supporting more planes in an airspace with a quick refresh rate. This document will describe each component of the ALU, including its inputs and outputs. Each component will be broken down into its fundamental logic, then built upon in combination with other logic to become fully functional.

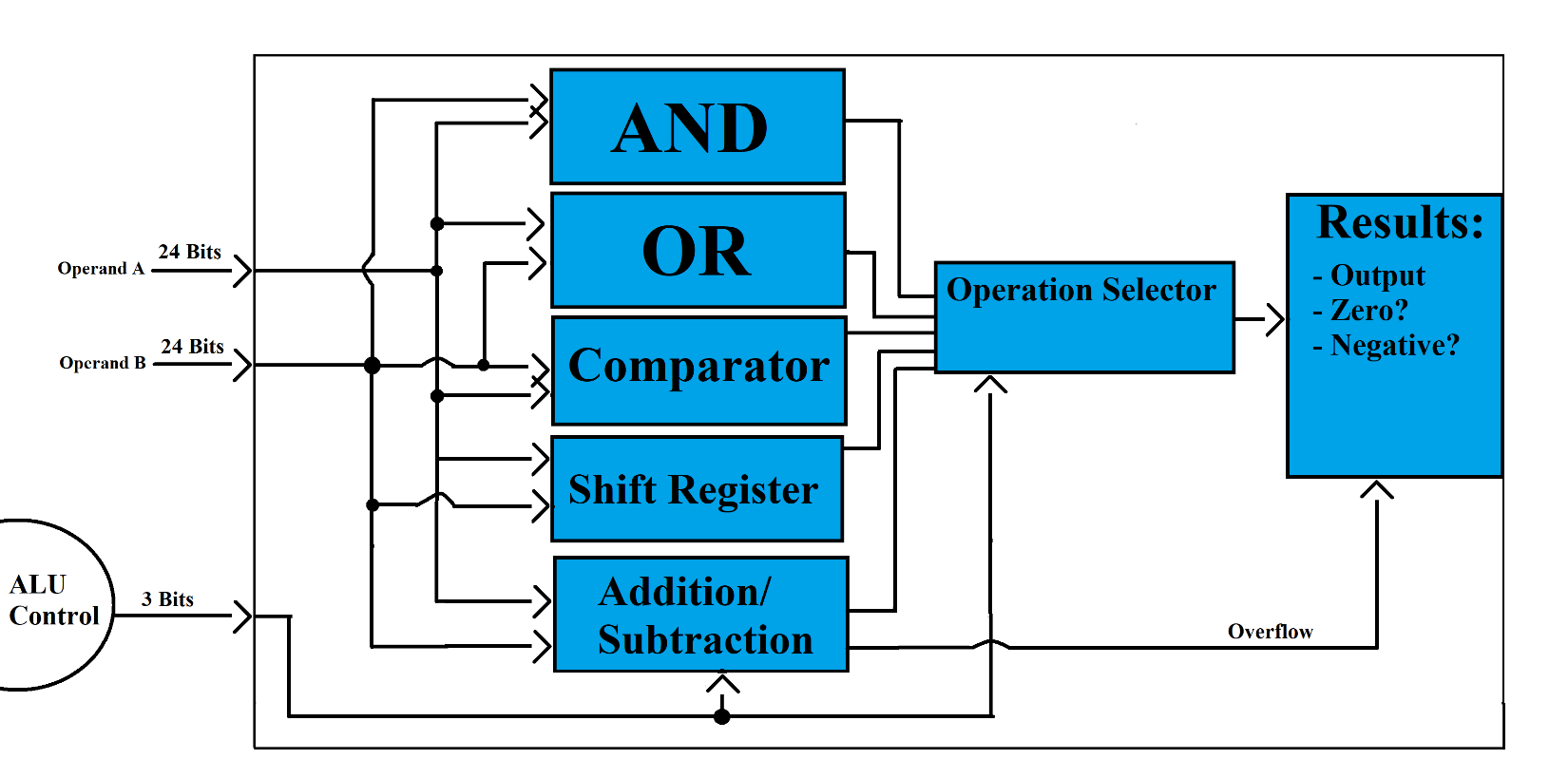
# High-Level Description:

# The ALU acts as the decision maker for the CPU. It must accept data from other systems and makes the necessary comparisons and decisions to choose what must happen next. This action can consist of commanding a multiplexer to pass different signals, sending a command outside of the CPU to another system, and more. For example, the ALU may receive data from the CRC circuit that it has finished running, and then must decide what to do next. More specifically, inside the microprocessor the ALU can tell the register file to store the data in its first address into the data memory. To make these decisions, the ALU must be able to perform addition, subtraction, comparisons, and must read and send data to registers. When reading instructions, the ALU must understand how to interpret any commands it receives. For example, the ALU Control will send an instruction regarding the operation the ALU must perform on its two operands. The ALU must interpret this instruction properly and provide outputs to downstream circuits.

# Inputs and Outputs:

The ALU will take in information to tell it what to perform, what to perform on, and where it should go. The ALU receives an instruction from the opcode that tells the ALU which address the data is located and where its result should go. More specifically, it contains the destination address, 2 addresses to “pickup” data from, and the operation. The ALU then retrieves its operands, and with the operation it performs its task and outputs the result. Two condition bits may also be used by other systems to further interpret the result.

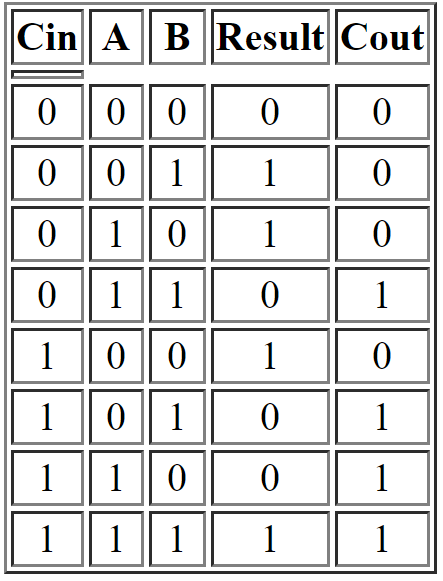
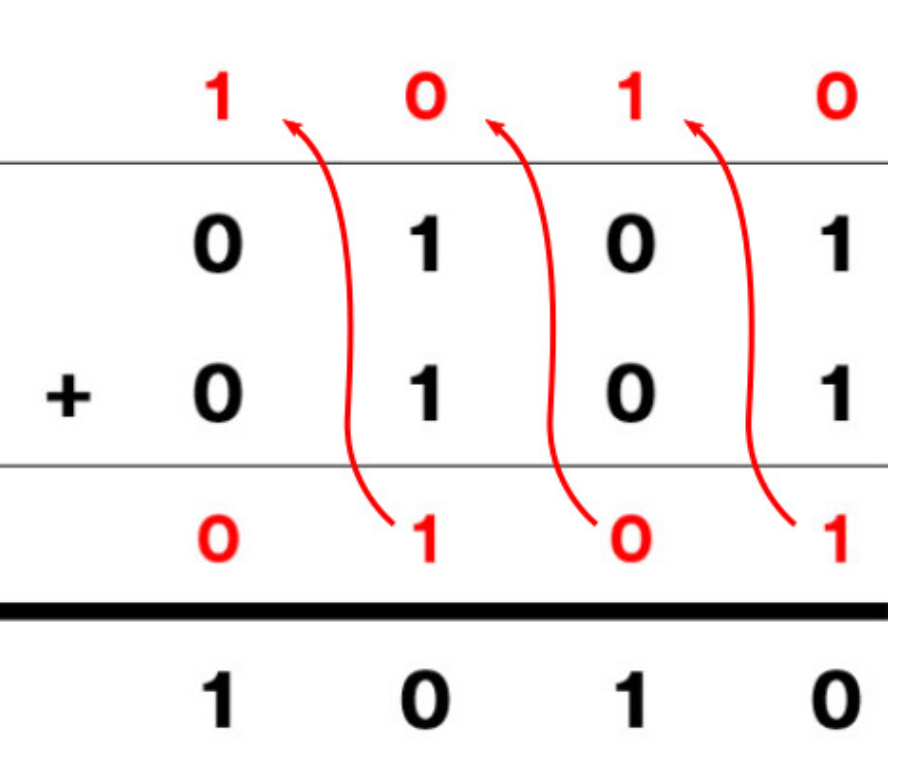
# ALU Interfaces:



The ALU takes in 2 operands and performs every operation at the same time. Then, the ALU control decides which operation it wants to output through the operation selector, then outputting that result and its features.

## Addition:

Addition consists of 32 1-bit full adders, which are logic systems that perform binary addition. The full adder adds three binary addends, A, B, and Carry In and produces a sum and a Carry Out. The Carry Out is sent to the next full adder (a binary digit above the previous as the Carry In) and continues the cycle.



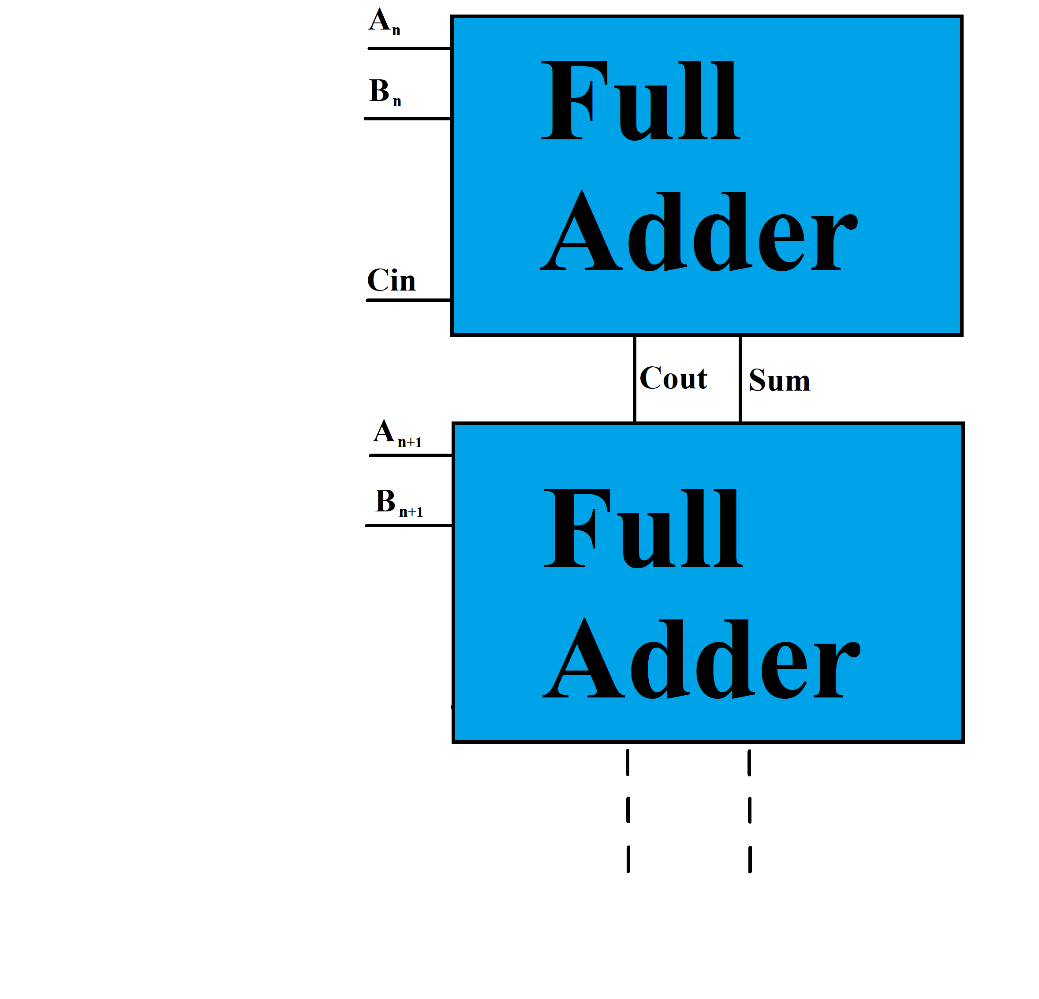
**Cin**

**A**

**B**

**Cout**

**Result**

Carries can be seen in the example above of 5 + 5. When two 1’s are added, a 0 is the result and a carry is sent to the next binary digit (if 3 1’s are in succession, a 1 AND a carry are produced). This continues until all of the operands’ bits have been added.

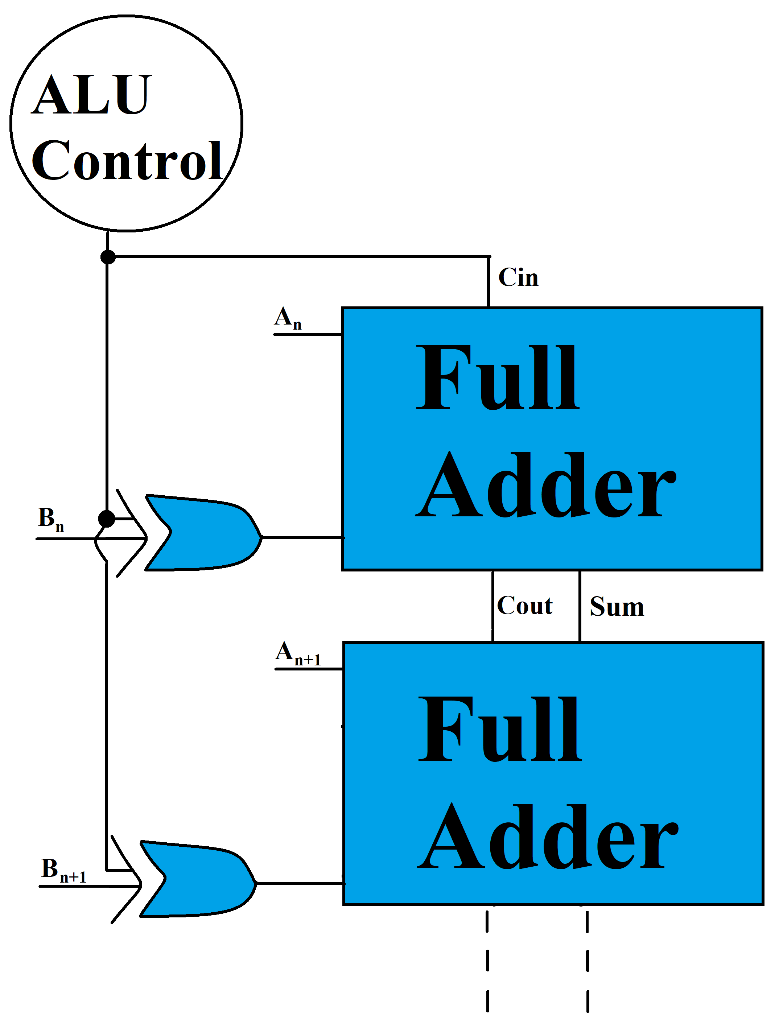
In logic, each binary addition would take place in individual full adders, with two shown to the right.

The bits from each operand are paired by their sequence in the data, then sent to the full adder corresponding to their sequence.

If there is a carry at the last full adder, then an overflow is created and the carry is placed on the first bit that determines the sign of the result. The solution to this issue will be explained in the results section.

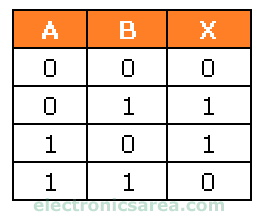
## Combining Addition and Subtraction:

Subtraction will use the same full adder system as addition but requires more logic to switch between them. In general, the system will “add” a negative number, which turns into subtraction:

A + (-B) = A + ~B + 1

The second equation is the binary form of adding a negative, called two’s compliment. It inverts the value of the second bit and adds 1 to the total. To apply this to the full adders, an XOR gate is connected to the B operand and the ALU Control, to change the value based on if addition or subtraction is occurring. This logic is added to the previous interface to finish the design.

Below is the truth table for an XOR gate:



For this table, “A” is the data, “B” is the ALU Control’s value, and “X” is the result. If the Control wants subtraction, then the data needs to be flipped. It does this by sending a 1 to an XOR gate, inverting the bit’s value. The Control’s 1 will also be the “carry-in” needed to perform subtraction. So, if a 1 is sent, then the data’s bit gets flipped and a 1 is added to the total bit. If a 0 is sent, then the data stays the same and addition occurs, and no carry-in is added.

## Boolean AND:

A 24-bit Boolean AND must produce a value of 1 whenever two operands are equal. Each pair of bits comes from either operand, in order of sequence. Many AND gates will not always produce the correct response because it produces a 0 when both operands are 0. In order to produce a high value whenever both bits are equal, XNORs are needed. A series of 6 logic cycles will take place, the first being 12 XNOR gates, the next taking those result into 6 XNOR gates, following with 3, then a series of 2 more. This will allow every bit to connect into one system.

## Boolean OR:

A 24-bit Boolean OR produces a value of 1 whenever a pair of bits are opposite. Each pair of bits comes from either operand, in order of sequence. 24 OR gates in parallel will produce the correct output. A series of 6 logic cycles will take place, the first being 12 OR gates, the next taking those result into 6 OR gates, following with 3, then a series of 2 more. This will allow every bit to connect into one system.

## Comparator:

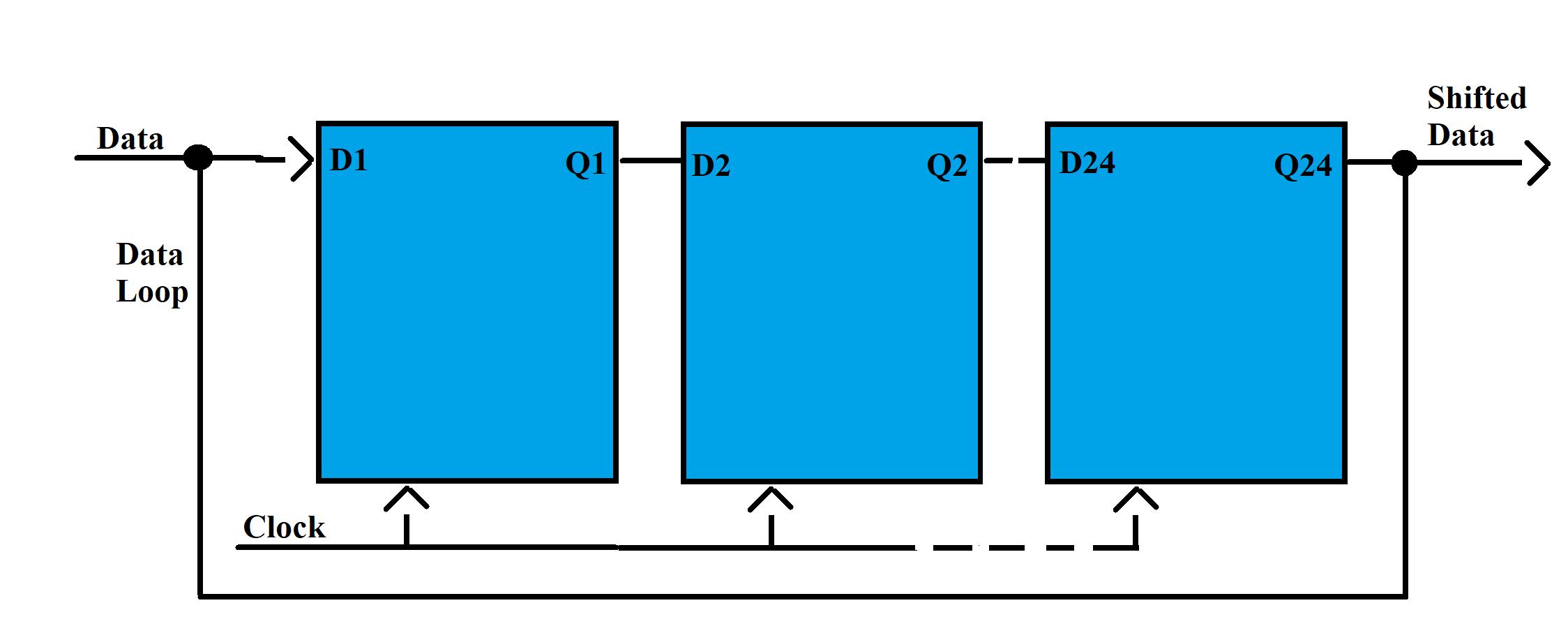
A comparator is an operation that can detect if one operand is greater than, less than, or equal to the other. It does this by running 3 different functions that each produce a different output, where the function that produces a 1 is the result. To see if every bit is equal, 24 XNORs are used on each pair of bits. These pairs are produced by selecting bits from each operand sequentially. The XNOR will produce a 1 is the bits are equal, so with a 24-bit AND gate, both operand’s equality can be seen.

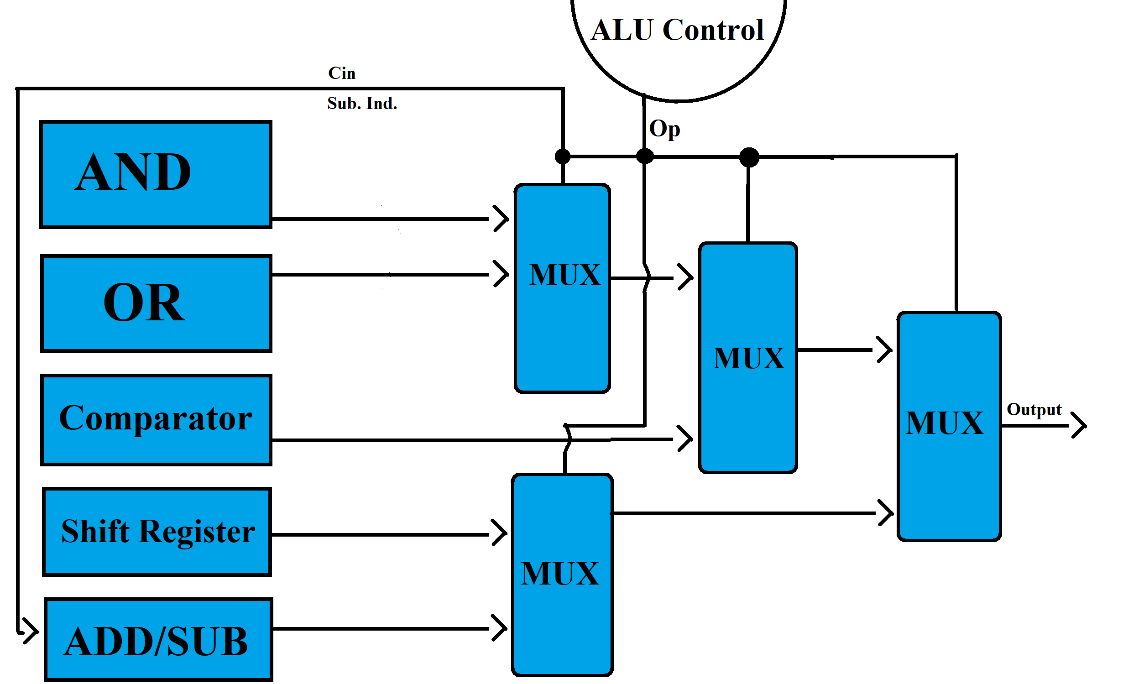
To detect an inequality, the function must detect differences between the operands. The function will compare bits from both operands from most significant to least significant. Then, when it finds an inequality, the operand with the higher value is determined to be greater than the other bit. This will be outputted as a variable between the 3 possible outputs: greater than, less than, or equal to. Only one of these outputs will receive a 1, with the others staying as 0, which produces a 3-bit output.

## Shift Register:

A shift register takes in an operand and manipulates it by cycling either left or right. To do this, the shift register takes the bit closest to the direction it wants to shift and moves it to the opposite end of the data. An example of this can be seen to the right.

In this example, a 1-bit right shift has occurred, which took the right-most bit and moved it to the other end of the data. This is how a shift occurs. In logic, flip flops are used to store and move the data. Flip flops maintain an inputted value until another input changes its value. When an input from the clock or another system is received, the flip flop will output its bit and input the next one. This allows the flip flops to act as temporary storage, where data can be broken down and manipulated.

24 flip flops are needed to store one operand. They will be connected to each other, receiving inputs from the previous flip flop or the data, and pulses from an outside system that will control how many times the system must shift. The flip flops will also loop the data, to prevent the data from being erased if needed. A gate-like system will only let the data loop if it receives a 1 from an outside system.



## Operation Selector:

Incorporates 4 multiplexers to select which operation (AND, OR, addition, subtraction, comparison, shift) it will to perform. Each multiplexer will pass 1 of the 2 operation outputs in a tiered arrangement so only one output passes through the ALU. The 3-bit data received from the ALU Control is what commands which output to send through.

## ALU Control:

Decodes instructions received from the data memory and commands the ALU on what operation to perform. The ALU Control sends 3 bits of data, that chooses which operation it wants outputted, and initially decides between addition and subtraction. The data can create 8 different outputs, however only 5 are needed to choose every operation. The system chooses its operation by interpreting instructions from the data memory in the microprocessor. This system is outside of the ALU and will not be designed in this paper.

## Results:

The output of the ALU is determined by what operation occurs. If an AND or OR operation is performed, their outputs are simply exported directly. If addition or subtraction occurs, then other features of the result need to be exported with the result: if the result is zero, if the result is negative, and if there was an overflow.

If the result is zero, the direct output cannot display it correctly. Therefore, another system is needed to determine if it is zero. This can be seen when every bit is zero and can be detected by a 24-bit OR gate and a NOT gate, that only outputs a 1 when every bit is 0. This works because if any bit in the OR gate is a 1, then the OR gate would produce a 1 that would then be flipped.

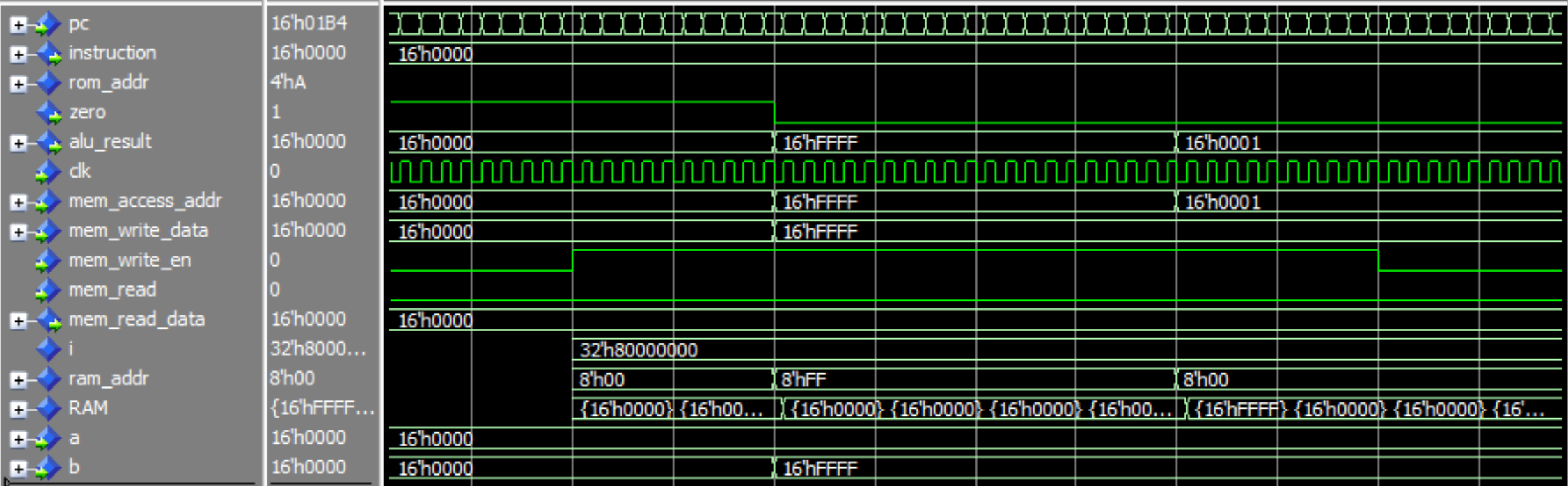
If the result is negative, it is expressed in the first bit in the output. This value can be altered, though, by carry bits, called an overflow. Addition starts with the last bit and ends with the first bit. As stated previously, if a carry is present on the last bit, then an overflow has occurred and the sign of the result isn’t correct. This can be detected with logic at the last full adder, that detects if the input’s sign isn’t the same as the output’s. If so, then an overflow has occurred, and the bit is simply flipped.

## Methods:

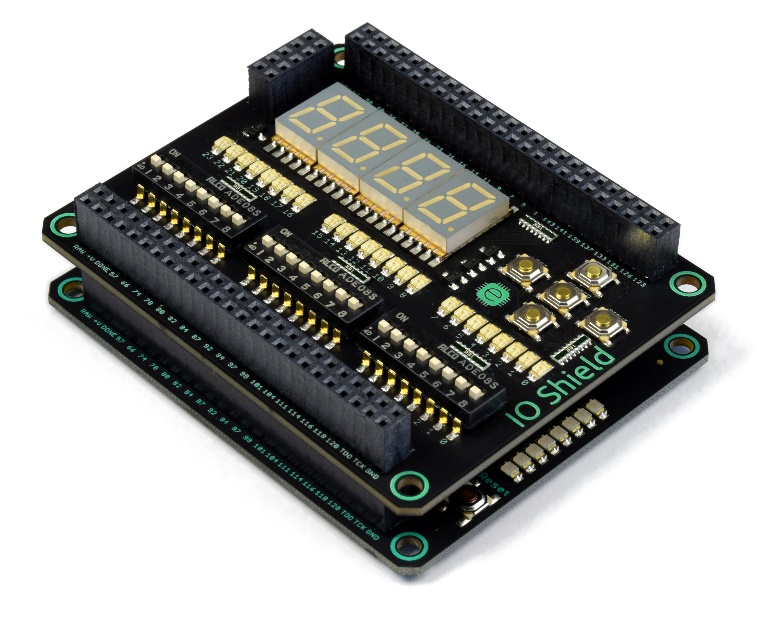
As stated previously, this project will focus on planning, implementing and validating a custom ALU for use in the microprocessor of an error-correcting system. Planning the ALU will require multiple programs and systems to correct or manipulate the code to be used on an Alchitry Au board, a Field Programmable Gate Array (FPGA) digital logic board.

The first iterations of code will be designed in ModelSim, creating test benches that can simulate running the code as if it was on the FPGA. This will speed up the process of creating new iterations of code and reduce the risk of damaging the FPGA due to critical errors. On a computer, Vivado will manipulate the validated code so that it can enter Alchitry Loader, a software that will communicate with the FPGA via a USB-C cable. The ALU will also output values onto a separate display from the computer. Using a bread board, LEDs, resistors and other wiring, a prototype of the ALU’s display system will be created. This system will help make outputs more noticeable; for example, an LED could light up when the ALU receives a message that an error is present.

Once the first version of the ALU has proven successful, the remaining task will be validating, or error-correcting, the code and finalizing the interface. ModelSim can break down the code into its waveforms over time, as shown below. This will be a form of discovering errors in the system.



In this example, the data 16’FFFF was stored in the address 16’FFFF.

GitHub will also aid in the validating process, by making iterations of code publicly accessible. This allows the code to be tested and reviewed on many platforms and keeps a detailed timeline of objectives relative to programming.

Once the ALU has been fully created, an interface will be prototyped through a breadboard with components that can act as inputs and outputs of the system. This prototype will then be implemented into a shield for the ALU similar to the one on the right. The shield is the final version of the interface, that allows the ALU to be experimented with outside of a testbench.

### Resources:

<https://minnie.tuhs.org/CompArch/Tutes/week02.html>

<https://learn.digilentinc.com/Documents/281>

<https://study.com/academy/lesson/arithmetic-logic-unit-alu-definition-design-function.html>

<https://en.wikipedia.org/wiki/Two%27s_complement>

<https://www.geeksforgeeks.org/digital-logic-shift-registers/>